Homework2 Report

MIAO QI
A0159327X

Task0:

Question: *List down and explain the differences between the Android VM and actual Android devices. (10 Marks)*

Answer:

1) Bootloader

Explanation: In actual Android devices, bootloader is a low level code that loads an operating system. In the Android VM, virtual machine (in this lab, it means Virtual Box) performs the task of loading operating system.

Another thing is, in actual Android devices, bootloader is usually locked, which means any attempt to flash the installed OS will be denied by the bootloader. Some vendors choose to permanently lock the bootloader. But some choose not to do so and instead, users doing this will void the warranty and completely wipe out personal data. In Android VM, especially in this lab, bootloader is unlocked and we have installed a custom recovery OS.

2) Recovery OS

Explanation: In actual Android devices, recovery OS is some kind of special OS such as TWRP, ClockworkMod. But in Android VM, especially in out lab, we use Ubuntu 15.10 as the recovery OS.

3) Architecture

Explanation: In actual Android devices, it can be 32-bit architecture or 64-bit architecture. In out lab, our Android VM is Android-x86 architecture.

4) Naming

Explanation: The difference on naming results from naming. For example, the daemon process Zygote, can be named zygote32 or zygote64, for 32-bit architecture and 64-bit architecture, and the app process, can be named app_process32 or app_process64.

In our lab, since the Android VM is Android-64 bit architecture, the names are all based on 32 bit, such as app_process32 and Zygote32.

5) Image file loaded

Explanation: During the process of init, all rc script files are stored in an image file. In our lab, it is named ramdisk.img on our Android-x86 VM. On actual Android devices, these script files are inside boot.img, which contains ramdisk.img and the kernel.

6) Others

Explanation: There are some other difference between actual Android devices and an Android VM. The VM has more limits. There is limited support for VM to determine SD card insert/eject, and for determining battery charge level and AC charging state, and for placing for receiving actual phone calls, and so on. These functionalities may be simulated via some approaches on Android VM, but it will be much more harder than in a real device.

Task1: (40 Marks)

1. Question: *Identify which component in the signature verification process is problematic and explain why is it so. (10 Marks)*

Answer:

As is shown in the Figure1.1, The problematic thing is that **we have write permission on the certificate file**, which lies on /android/system/etc/security/otacerts.zip. It is because the Android allow users to import some certificates or may be just a “manual vulnerability” created by our TA or professor.

Figure1.1 “w” permission on certificate file cause vulnerability

So by replacing otacerts.zip with our own public key file wrapped in a zip, we can make the recovery OS trust our OTA file by decrypt the file using the fake certificate created by us.

2. Question: *Show and explain how the OTA update service can be made to verify our custom ota file. (10 Marks)*

Answer:

Signature verification is done using public key cryptography where the public key is used to verify the signature generated using the corresponding private key. The recovery OS verifies the signature via certificate file in /android/system/etc/security/otacerts.zip. So as shown in the previous question, since we have write permission on the certificate file otacert.zip, <u>we can replace it with our own public key</u>**.** The steps are as following

1) In Mobi, as shown in Figure 1.2.1, I generate my private key (pk8), public key (certificate) of RSA using openssl.

```
seed@MobiSEEDUbuntu:~/Desktop/Task1$ openssl genrsa -out personal.pem 2048
Generating RSA private key, 2048 bit long modulus
....................+++
....................+++
e is 65537 (0x10001)
```

```
seed@MobiSEEDUbuntu:~/Desktop/Task1$ mv personal.pem private.pem
seed@MobiSEEDUbuntu:~/Desktop/Task1$ openssl req -new -key private.pem -out requ
est.pem
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:cn
State or Province Name (full name) [Some-State]:zj
Locality Name (eg, city) []:hz
Organization Name (eg, company) [Internet Widgits Pty Ltd]:zju
Organizational Unit Name (eg, section) []:m
Common Name (e.g. server FQDN or YOUR name) []:m
Email Address []:m

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

```
seed@MobiSEEDUbuntu:~/Desktop/Task1$ openssl x509 -req -days 9999 -in request.pe
m -signkey private.pem -out certificate.pem
Signature ok
subject=/C=cn/ST=zj/L=hz/O=zju/OU=m/CN=m/emailAddress=m
Getting Private key
seed@MobiSEEDUbuntu:~/Desktop/Task1$ openssl pkcs8 -topk8 -outform DER -in priva
te.pem -inform PEM -out key.pk8 -nocrypt
```

Figure 1.2.1 Generate my own private key, public pey and pk8

2) Use private key and certificate to sign ota.zip using

~/android/apktool/signapk.jar. This is shown in Figure 1.2.2.

```
seed@MobiSEEDUbuntu:~/android/apktool$ java -jar signapk.jar -w certificate.pem
key.pk8 otabackup.zip ota.zip
```

Figure 1.2.2 Sign ota.zip with private key(pk8)

Especially, here we need to use –w parameter to sign all the files in the package.

3) Pack my own public key (certificate) as otacerts.zip, transfer it into android file system via scp command, and use the new otacerts.zip to replace the old one in /android/system/etc/security/ .

```
seed@MobiSEEDUbuntu:~/android/apktool$ zip otacers.zip certificate.pem
  adding: certificate.pem (deflated 31%)
seed@MobiSEEDUbuntu:~/android/apktool$ mv otacers.zip otacerts.zip
```

```
seed@MobiSEEDUbuntu:~/android/apktool$ scp otacerts.zip  seed@10.0.2.4:/home/see
d/ota/otacert.zip
seed@10.0.2.4's password:
otacerts.zip                                  100%  996     1.0KB/s   00:00
```

```
seed@recovery:~$ cp /home/seed/ota/otacert.zip /android/system/etc/security/otac
erts.zip
seed@recovery:~$ ls -l /android/system/etc/security/otacerts.zip
-rw-rw-rw- 1 root root 996 Oct 21 08:27 /android/system/etc/security/otacerts.zi
p
seed@recovery:~$
```

Figure 1.2.3 Replace /android/system/etc/security/otacerts.zip with our own one

4) Flush the ota.zip. Successfully verify the signature

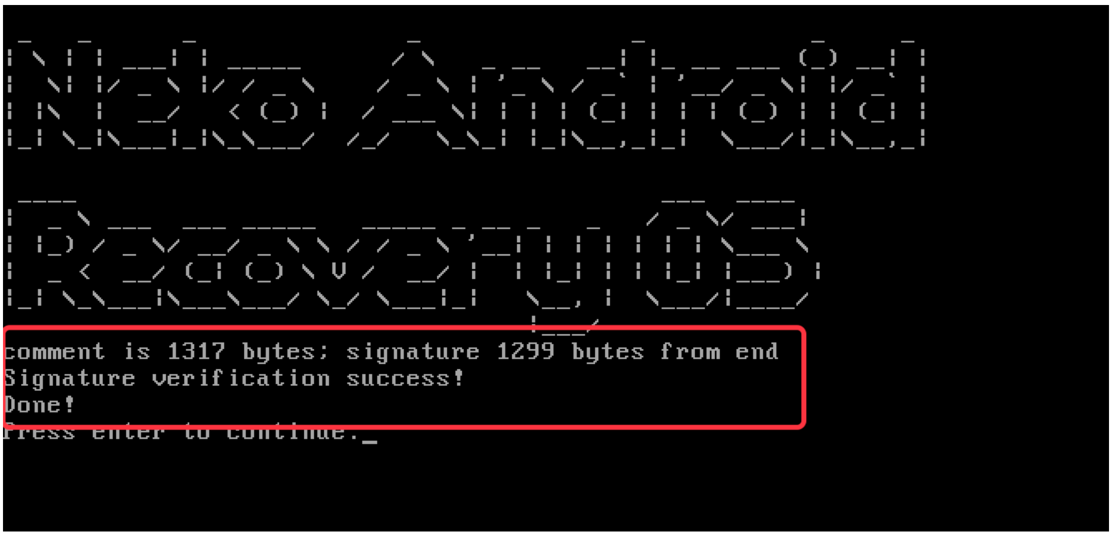

Figure 1.2.4 Successfully verify the signature

*3. **Question:** Show the structure of OTA file. (10 Marks)*

Answer:

```
seed@MobiSEEDUbuntu:~/Desktop/Task1$ unzip -l ota.zip
Archive:  ota.zip
signed by SignApk
  Length      Date    Time    Name
---------  ---------- -----   ----
      185  2016-10-21 07:46   META-INF/com/google/android/update-binary
       27  2016-10-16 05:32   system/dummy.sh
     1180  2016-10-21 07:46   META-INF/com/android/otacert
      268  2016-10-21 07:46   META-INF/MANIFEST.MF
      321  2016-10-21 07:46   META-INF/CERT.SF
     1293  2016-10-21 07:46   META-INF/CERT.RSA
---------                     -------
     3274                     6 files
seed@MobiSEEDUbuntu:~/Desktop/Task1$
```

Figure 1.3.1 OTA Structure

My OTA structure is shown in Figure 1.3.1. I think the most important file is dummy.sh and update-binary and the package has been signed. I mark them in red lines.

dummy.sh is in system/ folder, and update-binary is in META-INF/com/google/android folder. Some essential files for signature are generated by signapk.jar. Some unnecessary folders are not included in my ota.zip.

The content of dummy.sh is a simple command: echo hello > /system/dummy, which write the string hello into /system/dummy.

```
find /tmp -name "dummy.sh" -exec cp {} /android/system/xbin/dummy.sh \;
chmod +x /android/system/xbin/dummy.sh
sed -i "/return 0/i /system/xbin/dummy.sh" /android/system/etc/init.sh
```

Figure 1.3.2 update-binary

As is shown in Figure 1.3.2, update-binary contains three commands. The first is to find "dummy.sh" and copy it into android file system. Since we know it will be put into somewhere in /tmp, here I combine the “find” and “cp” commands to simplify the shell program. The second command is insert “/system/xbin/dummy.sh” before “return 0” in init.sh. So before init.sh terminates, it will execute our dummy.sh and write dummy into /system as root privilege. And the third line is to make sure we have “x” privilege on dummy.sh

4. Question: *Show that the dummy file is created within the Android system. (10 Marks)*

Answer:

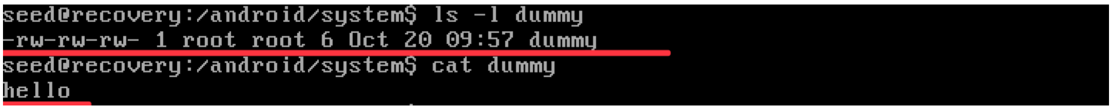

Figure 1.4 OTA Structure

As is shown in Figure 1.4, dummy has already been written in /android/system folder and the string ‘hello’ has been successfully echoed in dummy.

Task2: (40 Marks)

1. Question: *Show the structure of OTA file. (10 Marks)*

Answer:

```
seed@MobiSEEDUbuntu:~/Desktop/Task2/ota$ unzip -l ota.zip
Archive:  ota.zip
signed by SignApk
  Length      Date    Time    Name
---------  ---------- -----   ----
      162  2016-10-21 07:46   META-INF/com/google/android/update-binary
     5408  2016-10-21 07:46   system/compiled_binary
     1180  2016-10-21 07:46   META-INF/com/android/otacert
      275  2016-10-21 07:46   META-INF/MANIFEST.MF
      328  2016-10-21 07:46   META-INF/CERT.SF
     1293  2016-10-21 07:46   META-INF/CERT.RSA
---------                     -------
     8646                     6 files
seed@MobiSEEDUbuntu:~/Desktop/Task2/ota$
```

Figure 2.1.1 OTA Structure

My OTA structure is shown in Figure 2.1.1 The most important file is compiled_binary compiled by NDK and update-binary and files for signature verification. I mark them in red lines.

Like in Task1, I put the compiled_binary in system/ folder, and update-binary META-INF/com/google/android folder.

```
mv /android/system/bin/app_process32 /android/system/bin/app_process_original
find /tmp -name "compiled_binary" -exec cp {} /android/system/bin/app_process32
\;
chmod +x /android/system/bin/app_process32
```

Figure 2.1.2 The update-binary file

As is shown in the Figure 2.1.2, the update-binary contains three commands. The first one is rename app_process32 as app_process_orinal. The second one is to copy compiled_binary to /android/system/bin/app_process32. Here as in Task1, I still combine the find and cp command to achieve my goal. The third one is to make sure app_process32 can be executed with proper privilege.

Our fake app_process32(which actually is compiled_binary) will write to /system/dummy2 and then call the real app_process32(which now is app_process_original).

2. Question: *Show that the dummy file is created within the Android system. (10 Marks)*

Answer:

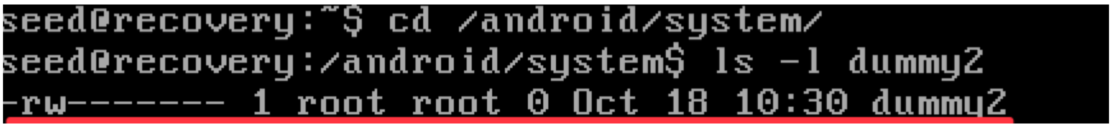

Figure 2.2 dummy2

As shown in Figure 2.2, after update the system via flushing ota.zip and restart the system, we have got dummy2 in /system as expected.

3. Question: *Highlight and explain the differences between the dummy files are created in Task 1 and Task 2. (20 Marks)*

Answer:

The most important difference is that the two dummy files in Task1 and Task2 are generated in different stages of booting process.

The dummy file in Task1 is generated in “Init” process. After the kernel is loaded, Init is created as the first user-space process. It is the starting point for all other processes, and it is running under the root privilege. Init initializes the virtual file system, detects hardware, then executes script file init.rc to configure the system. All the rc script files imported to init.rc are stored in an image file, named ramdisk.img on our

Android-x86 VM; The ramdisk.img file will be extracted into the memory while booting up. Making modifications directly on an image file is not very easy. Therefore, we only change the init.sh file in Task1 since the init.sh file is inside the /system folder, not
inside those image files.

The dummy file in Task2 is generated in app_process32. This is executed by "Zygote" process, which is launched by the Init process. "Zygote" is a daemon process, which executes the app process binary and has root privilege. Our fake app_process32 writes to /system/dummy2 and then call the real app_process32.

Another difference is, the dummy files in Task1 and Task2 have different privilege of owner/groups/others as shown in Figure 2.4. This is because the init.sh creates dummy by executing dummy.sh and app_process32 creates dummy2 by executing compiled binary of a C program. After generating files, init.sh and app_process32(or the daemon) assign the file different default privileges.

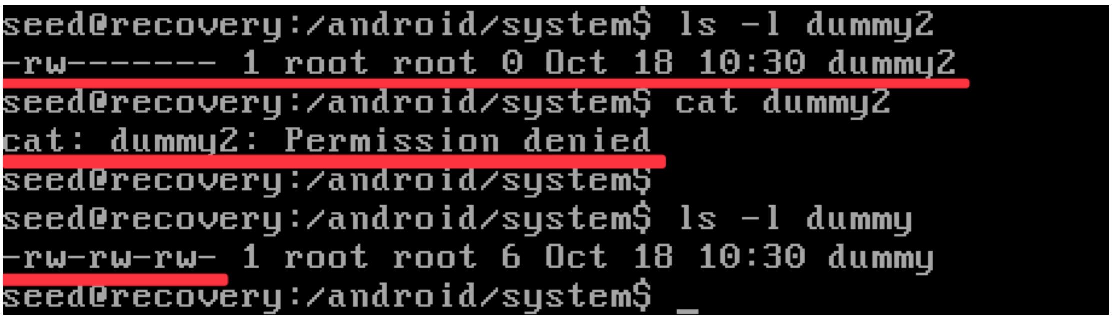

Figure 2.4: The dummy files in Task1 and Task2 have different privilege

Task3: (50 Marks)

1. Question: *Show that root shell can be obtained. (10 Marks)*

Answer: In this task, as shown in Figure 3.1, in update-binary, I use “mydaemon” to replace app_process32 and copy “mysu” into Android file system.

```
mv /android/system/bin/app_process32 /android/system/bin/fake_app_process32
find /tmp -name "mysu" -exec cp {} /android/system/bin/mysu \;
find /tmp -name "mydaemon" -exec cp {} /android/system/bin/app_process32 \;
chmod +x /android/system/bin/mysu
chmod +x /android/system/bin/app_process32
```

Figure 3.1.1: update-binary

After flush the ota.zip and reboot the Android VM, we can run mysu in Android terminal as not root privilege, and then we get the root shell. The root shell can be seen in Figure 3.1.2

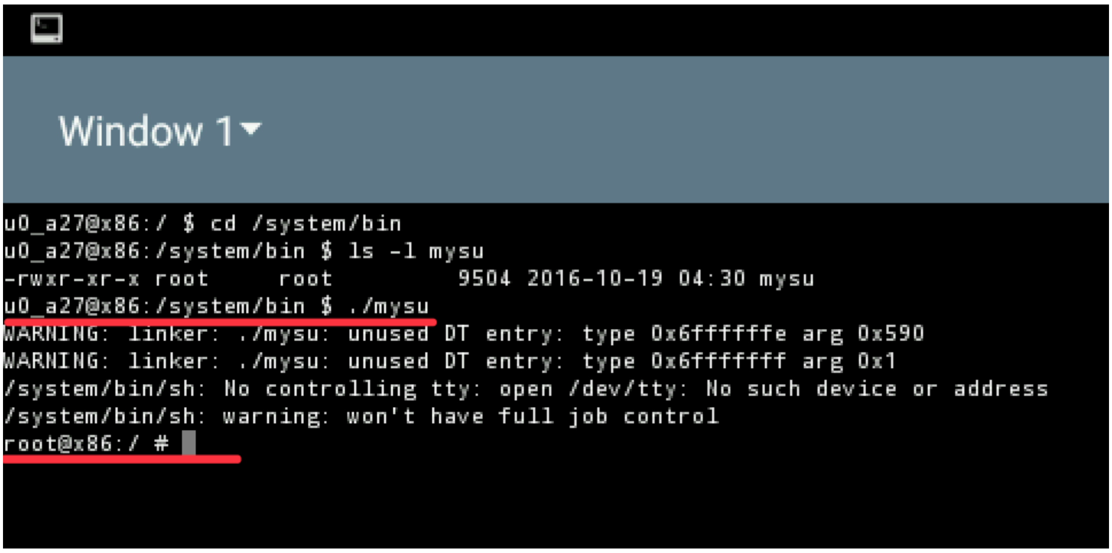

Figure 3.1.2: Get the root shell

2. Question: *Show that the client and shell processes share the same file descriptors. (10 Marks)*

Answer:

Firstly, I find pid of app_procees, mysu and the root shell /system/bin/sh (it is a child process of app_process) by using “ps” command.

```
root      1079  1070  9768   184   c13e041a b763d393 S /system/bin/app_process
system    1286  1070  1608416 114436 ffffffff b756c435 S system_server
```

```
u0_a27    2075  1070  1534024 79376 ffffffff b756be26 R jackpal.androidterm
u0_a27    2091  2075  10168  1608  00000000 b765e9a1 S /system/bin/sh
u0_a27    2109  2091  9768   828   c1479ae7 b771ae66 S ./mysu
root      2110  1079  10168  1572  00000000 b760b9a1 S /system/bin/sh
u0_a9     2118  1070  1512160 44204 ffffffff b756c435 S com.android.musicfx
```

Figure 4.1.1: Get pid of app_process, mysu, /system/bin/sh

The pid of the shell without root is 2091. This shell generates mysu, so mysu’s parent pid is 2091.

The pid of mysu is 2109, it connects to server(app_process) and server generates the root shell /system/bin/sh. The root shell’s parent pid is 1079, which is the pid of app_process.

So the process of pid 2109 and 2110 is exact the process of mysu and the shell process we are looking for. It is as my expected. Then I list the fd info of both the process of pid 2109 and 2110.

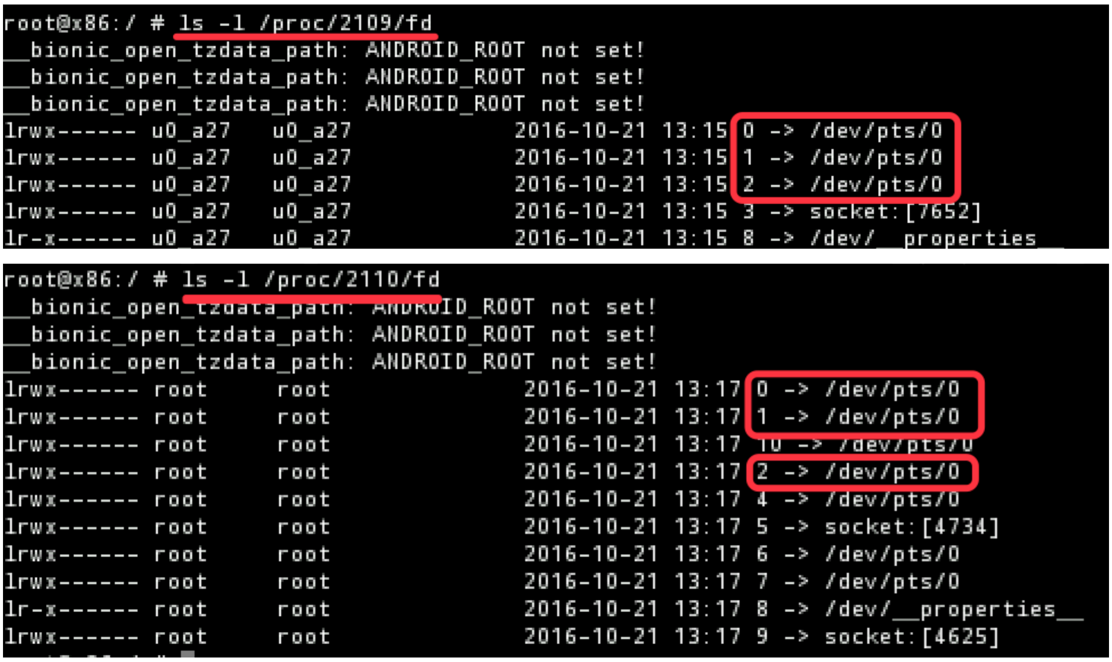

Figure 4.1.2: Process of 2109 and 2110 shares the same fd 0,1,2

As shown in Figure 4.1.2, I list the fd of the process 2109 and 2110. Via observation, **they both shared the fd 0, 1, 2, which confirms that they share the same file descriptors of IN, OUT, ERR.**

Another thing I notice is, mysu process has one socket fd, which is used to communicate with app_process. The shell process has two sockets fd, which are used to communicate with the server (app_process) and the client (mysu). These are all following our expectations.

3. Question: *Indicate where in the source files does the following actions happen (Filename, function name, and line number need to be provided* *in the answer: (30 Marks)*

- *Server launches the original app process binary*

Answer:

Filename: mydaemonsu.c

Function name: execve(argv[0], argv, environ), called in main()

Line number:

```
73  #define APP_PROCESS "/system/bin/app_process_original"
```

Line 73, it defines the path of original app process.

```
244  int main(int argc, char** argv) {
245      pid_t pid = fork();
246      if (pid == 0) {
247          //initialize the daemon if not running
248          if (!detect_daemon())
249              run_daemon();
250          }
251      else {
252          argv[0] = APP_PROCESS;
253          execve(argv[0], argv, environ);
254      }
255  }
256
```

Line 253, it calls execve() and so launch the original app process binary.

- *Client sends its FDs*

Answer:

Filename: called in mysu.c, realized in socket_util.c

Function name: send_fd() called by connect_daemon()

Line number:

```c
93  int connect_daemon() {
94
95      //get a socket
96      int socket = config_socket();
97
98      //do handshake
99      handshake_client(socket);
100
101     send_fd(socket, STDIN_FILENO);      //STDIN_FILENO = 0
102     send_fd(socket, STDOUT_FILENO);     //STDOUT_FILENO = 1
103     send_fd(socket, STDERR_FILENO);     //STDERR_FILENO = 2
104
```

Line 101-103 in mysu.c, call send_fd() to sends its fd.

```c
109 void send_fd(int sockfd, int fd) {
110     // Need to send some data in the message, this will do.
111     struct iovec iov = {
112         .iov_base = "",
113         .iov_len  = 1,
114     };
115
116     struct msghdr msg = {
117         .msg_iov        = &iov,
118         .msg_iovlen     = 1,
119     };
120
121     char cmsgbuf[CMSG_SPACE(sizeof(int))];
122
123     if (fd != -1) {
124         // Is the file descriptor actually open?
125         if (fcntl(fd, F_GETFD) == -1) {
126             if (errno != EBADF) {
127                 goto error;
128             }
129             // It's closed, don't send a control message or sendmsg will EBADF.
130         } else {
131             // It's open, send the file descriptor in a control message.
132             msg.msg_control    = cmsgbuf;
133             msg.msg_controllen = sizeof(cmsgbuf);
134
135             struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
136
137             cmsg->cmsg_len   = CMSG_LEN(sizeof(int));
138             cmsg->cmsg_level = SOL_SOCKET;
139             cmsg->cmsg_type  = SCM_RIGHTS;
140
141             *(int *)CMSG_DATA(cmsg) = fd;
142         }
143     }
```

```c
145     if (sendmsg(sockfd, &msg, 0) != 1) {
146 error:
147         PLOGE("unable to send fd");
148         exit(-1);
149     }
150 }
```

Line 109-150 in socket_util.c, realize the functionality of send_fd.

- *Server forks to a child process*

Answer:

Filename: mydaemonsu.c

Function name: child_process(), called in run_daemon(), called in main()

Line number:

```
187     int client;
188     while ((client = accept(socket, NULL, NULL)) > 0) {
189         if (0 == fork()) {
190             close(socket);
191             exit(child_process(client));
192         }
193         else {
194             close(client);
195         }
196     }
```

Line 191, call child_process(client) to fork a child process.

- *Child process receives client's FDs*

Answer:

Filename: called in mydaemonsu.c, realized in socket_util.c

Function name: recv_fd()

Line number:

```
143 int child_process(int socket){
144     //handshake
145     handshake_server(socket);
146
147     int client_in = recv_fd(socket);
148     int client_out = recv_fd(socket);
149     int client_err = recv_fd(socket);
```

Line 147-149 in mydaemonsu.c, call recv_fd(socket)

```c
52 int recv_fd(int sockfd) {
53     // Need to receive data from the message, otherwise don't care about it.
54     char iovbuf;
55
56     struct iovec iov = {
57         .iov_base = &iovbuf,
58         .iov_len  = 1,
59     };
60
61     char cmsgbuf[CMSG_SPACE(sizeof(int))];
62
63     struct msghdr msg = {
64         .msg_iov        = &iov,
65         .msg_iovlen     = 1,
66         .msg_control    = cmsgbuf,
67         .msg_controllen = sizeof(cmsgbuf),
68     };
69
70     if (recvmsg(sockfd, &msg, MSG_WAITALL) != 1) {
71         goto error;
72     }
73
74     // Was a control message actually sent?
75     switch (msg.msg_controllen) {
76     case 0:
77         // No, so the file descriptor was closed and won't be used.
78         return -1;
79     case sizeof(cmsgbuf):
80         // Yes, grab the file descriptor from it.
81         break;
82     default:
83         goto error;
84     }
85
86     struct cmsghdr *cmsg = CMSG_FIRSTHDR(&msg);
87
88     if (cmsg               == NULL                 ||
89         cmsg->cmsg_len    != CMSG_LEN(sizeof(int)) ||
90         cmsg->cmsg_level  != SOL_SOCKET            ||
91         cmsg->cmsg_type   != SCM_RIGHTS) {
92 error:
```

```c
93         LOGE("unable to read fd");
94         exit(-1);
95     }
96
97     return *(int *)CMSG_DATA(cmsg);
98 }
```

Line 52-98 in socket_util.c, realize the function of recv_fd().

- *Child process redirects its standard I/O FDs*

Answer:

Filename: mydaemonsu.c

Function name: dup2(), called in child_process()

Line number:

```
143 int child_process(int socket){
144     //handshake
145     handshake_server(socket);
146 
147     int client_in = recv_fd(socket);
148     int client_out = recv_fd(socket);
149     int client_err = recv_fd(socket);
150 
151     dup2(client_in, STDIN_FILENO);      //STDIN_FILENO = 0
152     dup2(client_out, STDOUT_FILENO);    //STDOUT_FILENO = 1
153     dup2(client_err, STDERR_FILENO);    //STDERR_FILENO = 2
```

Line 151-153, call dup2() to redirect its IN/OUT/ERR fds.

- *Child process launches a root shell*

Answer:

Filename: mysu.c

Function name: execve(shell[0], shell, NULL), called in main()

Line number:

```
136     //launch default shell directly
137     char* shell[] = {"/bin/sh", NULL};
138     execve(shell[0], shell, NULL);
139     return (EXIT_SUCCESS);
140 }
```

Line 137, define the shell path and parameter(NULL).

Line 138, launch the default shell.